Porting the Ginkgo Math Library to the oneAPI ecosystem

Terry Cojean, Hartwig Anzt, Fritz Goebel, Thomas Gruetzmacher, Aditya Kashi, Pratik Nayak, Tobias Ribizel, Yuhsiang M. Tsai
Steinbuch Centre for Computing (SCC)

Terry Cojean

Hartwig Anzt

Fritz Göbel

Thomas Grützmacher

Aditya Kashi

Pratik Nayak

Tobias Ribizel

Mike Tsai

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and the Helmholtz Impuls und VernetzungsfondVH-NG-1241.

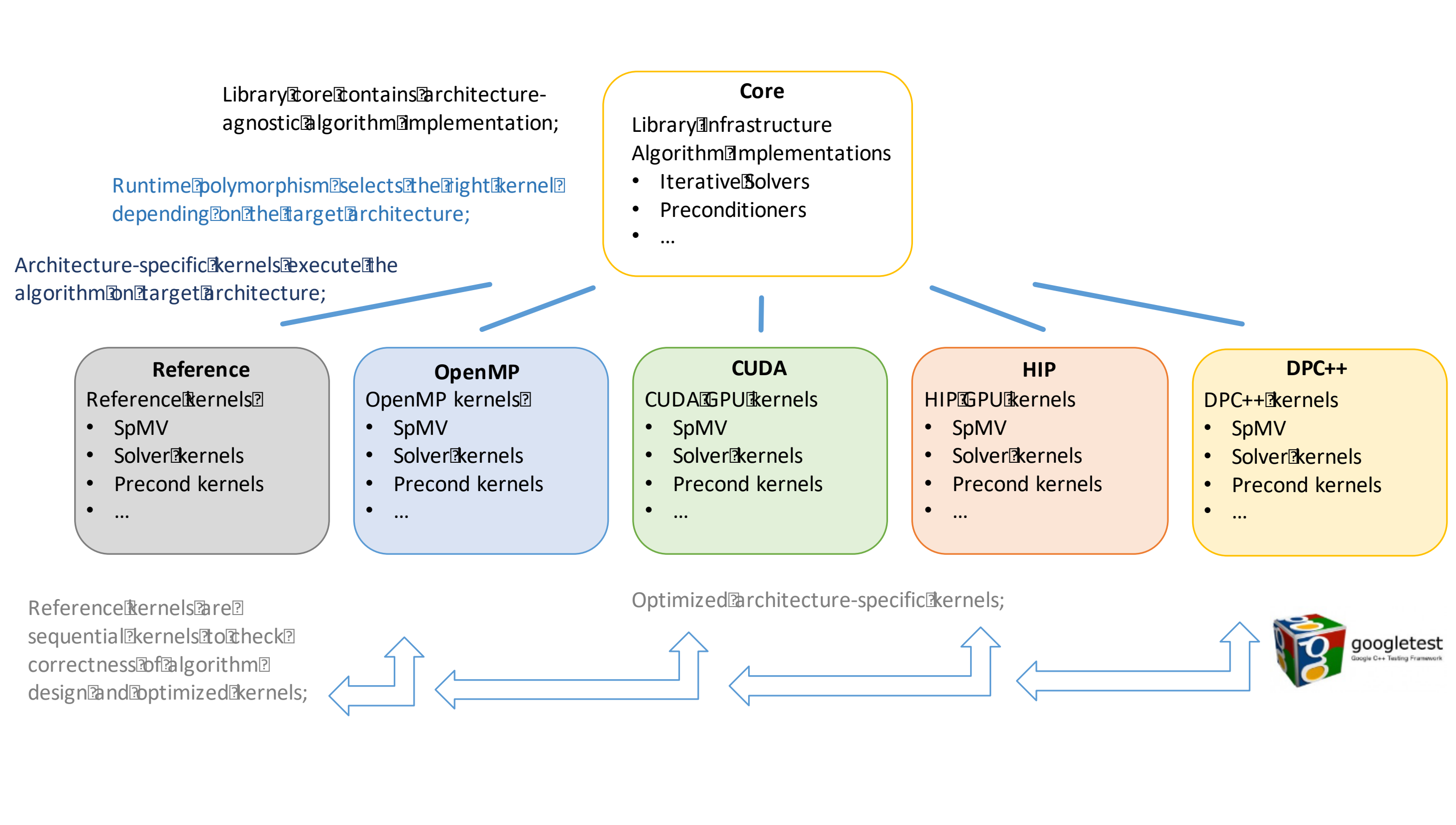
A Sustainable Open Source Math Software

Ginkgo

- **High performance sparse linear algebra**
  - Linear solvers, eigenvalue solvers;
  - Advanced preconditioning techniques
  - Decoupling of arithmetic precision (hardware-supported) and memory precision;
  - Linear algebra building blocks;
  - Extensible, sustainable, production-ready;

- **Open source, community-driven**
- Freely available (BSD License), GitHub, and Spack.
  - Part of the **E4S and xSDK software stack.**
  - Collaborative Effort:
  - Can be used from **MFEM**, and **deal.II**.

GPU-centric high performance sparse linear algebra ecosystem. Sustainable and extensible ecosystem with support for AMD GPUs, NVIDIA GPUs, and Intel GPUs.


Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

Architecture-specific kernels execute the algorithm on target architecture;

Core
Library infrastructure
Algorithm implementations
- Iterative Solvers
- Preconditioners
- …

Reference
Reference kernels
- SpMV
- Solver kernels
- Precond kernels
- …

OpenMP
OpenMP kernels
- SpMV
- Solver kernels
- Precond kernels
- …

CUDA
CUDA GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

HIP
HIP GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

DPC++
DPC++ kernels
- SpMV
- Solver kernels
- Precond kernels
- …

Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;

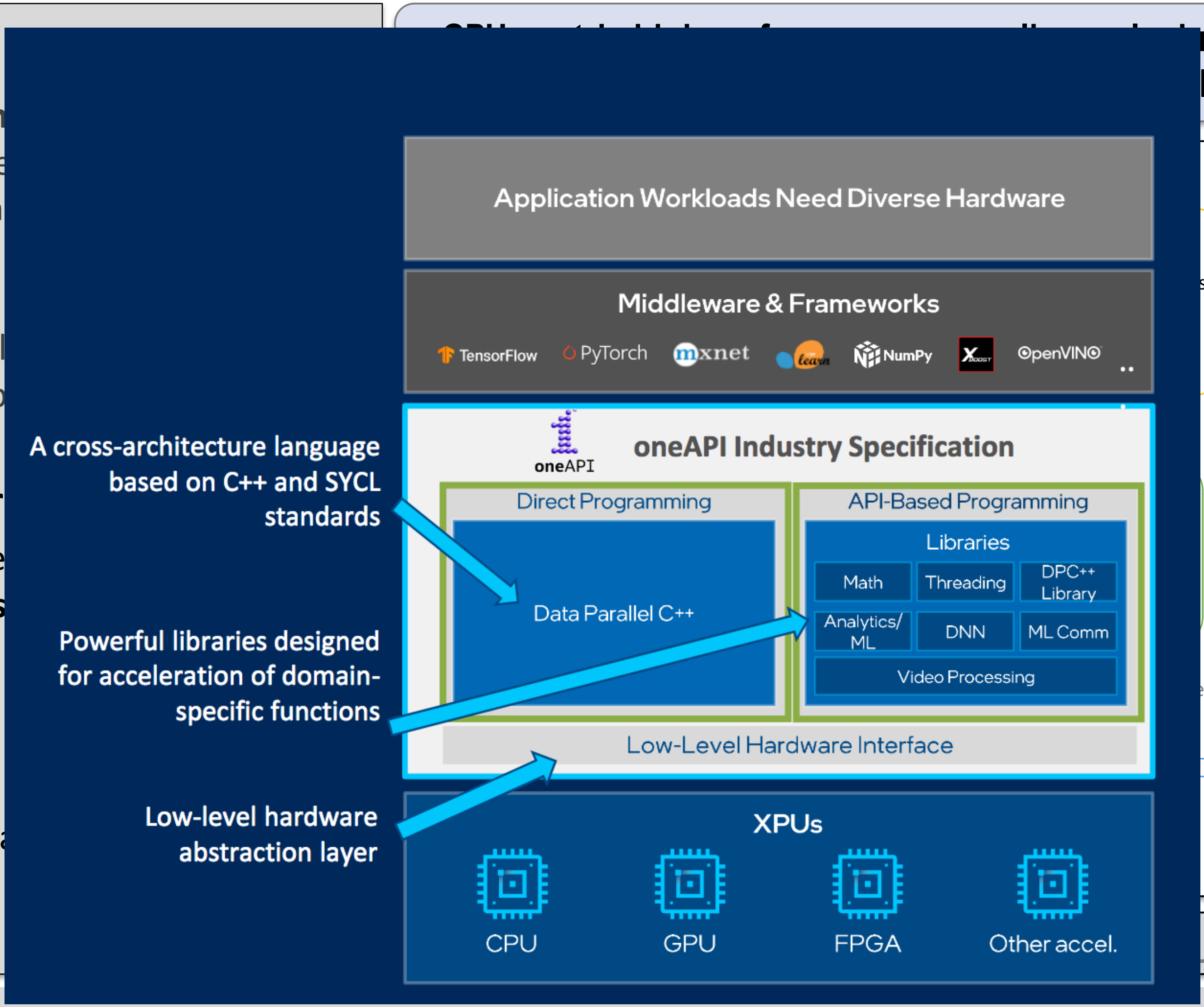
Optimized architecture-specific kernels;

https://ginkgo-project.github.io/

A Sustainable Open Source Math Software

Ginkgo

- **High performance sparse lin**
  - Linear solvers, eigenvalue
  - Advanced preconditionin
  - Decoupling of arithmetic supported) and memory
  - Linear algebra building bl
  - Extensible, sustainable, p
- **Open source, community-dr**
- Freely available (BSD License
  - Part of the **E4S and xSDK s**
  - Collaborative Effort:
  - Can be used from **MFEM**, a

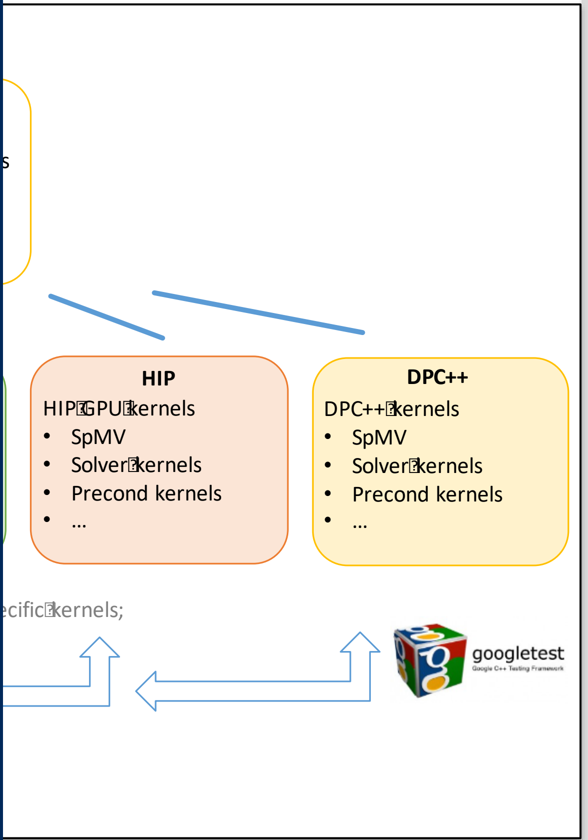


A cross-architecture language based on C++ and SYCL standards

Powerful libraries designed for acceleration of domain-specific functions

Low-level hardware abstraction layer

ra ecosystem. Sustainable and IDIA GPUs, and Intel GPUs.



ithub.io/

A Sustainable Open Source Math Software

Ginkgo

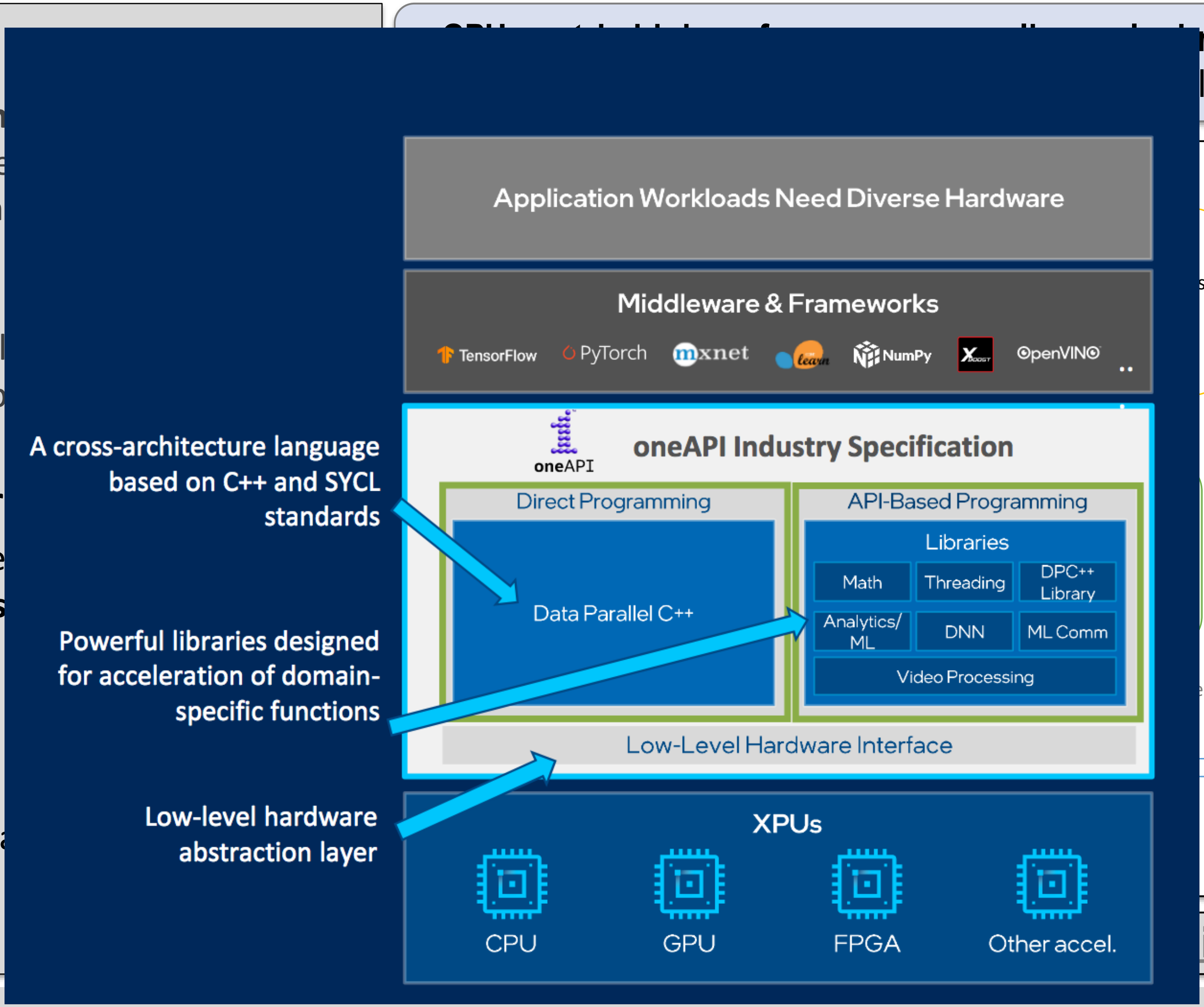
- **High performance sparse lin**
  - Linear solvers, eigenvalue
  - Advanced preconditionin
  - Decoupling of arithmetic supported) and memory
  - Linear algebra building bl
  - Extensible, sustainable, p

- **Open source, community-dr**
- Freely available (BSD License
  - Part of the **E4S and xSDK s**
  - Collaborative Effort:

KIT Karlsruhe Institute of Technology THE UNIVERSITY OF TENNESSEE KNOXVILLE

  - Can be used from **MFEM, a**

open source BSD

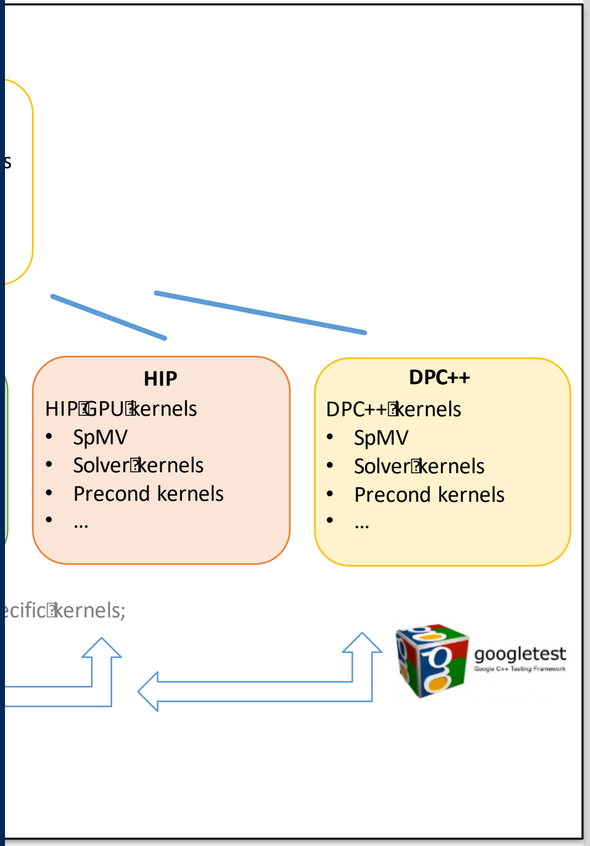


A cross-architecture language based on C++ and SYCL standards

Powerful libraries designed for acceleration of domain-specific functions

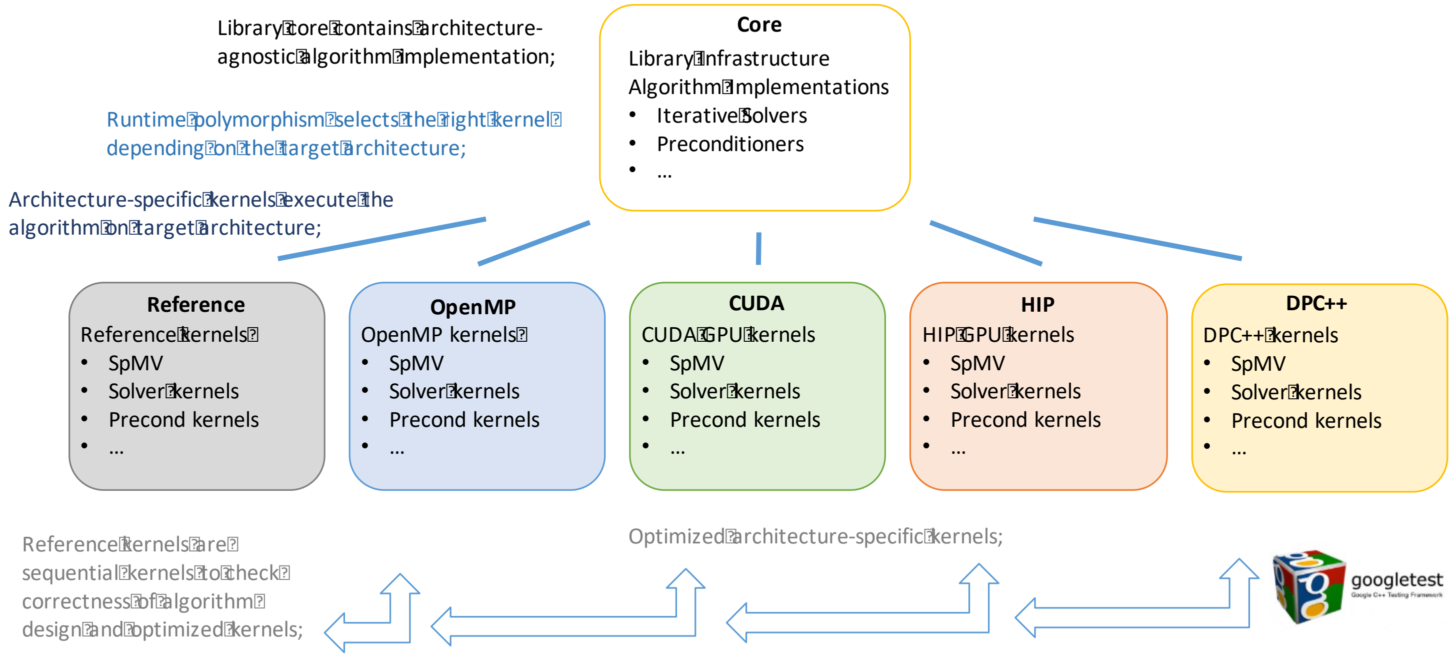
Low-level hardware abstraction layer

ra ecosystem. Sustainable and IDIA GPUs, and Intel GPUs.



ithub.io/

Ginkgo's plumbing between backends and algorithms



Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

Architecture-specific kernels execute the algorithm on target architecture;

Core

Library infrastructure

Algorithm implementations

- Iterative Solvers
- Preconditioners
- …

Reference

Reference kernels

- SpMV
- Solver kernels
- Precond kernels
- …

OpenMP

OpenMP kernels

- SpMV
- Solver kernels
- Precond kernels
- …

CUDA

CUDA GPU kernels

- SpMV
- Solver kernels
- Precond kernels
- …

HIP

HIP GPU kernels

- SpMV
- Solver kernels
- Precond kernels
- …

DPC++

DPC++ kernels

- SpMV
- Solver kernels
- Precond kernels
- …

Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;

Optimized architecture-specific kernels;

Operation

Run(OmpExecutor...)
Run(CudaExecutor...)
…

Executor

Run(Operation)
Allocate(size, …)
Free(ptr)
Copy(ptr, OtherExec)
…

CudaExecutor

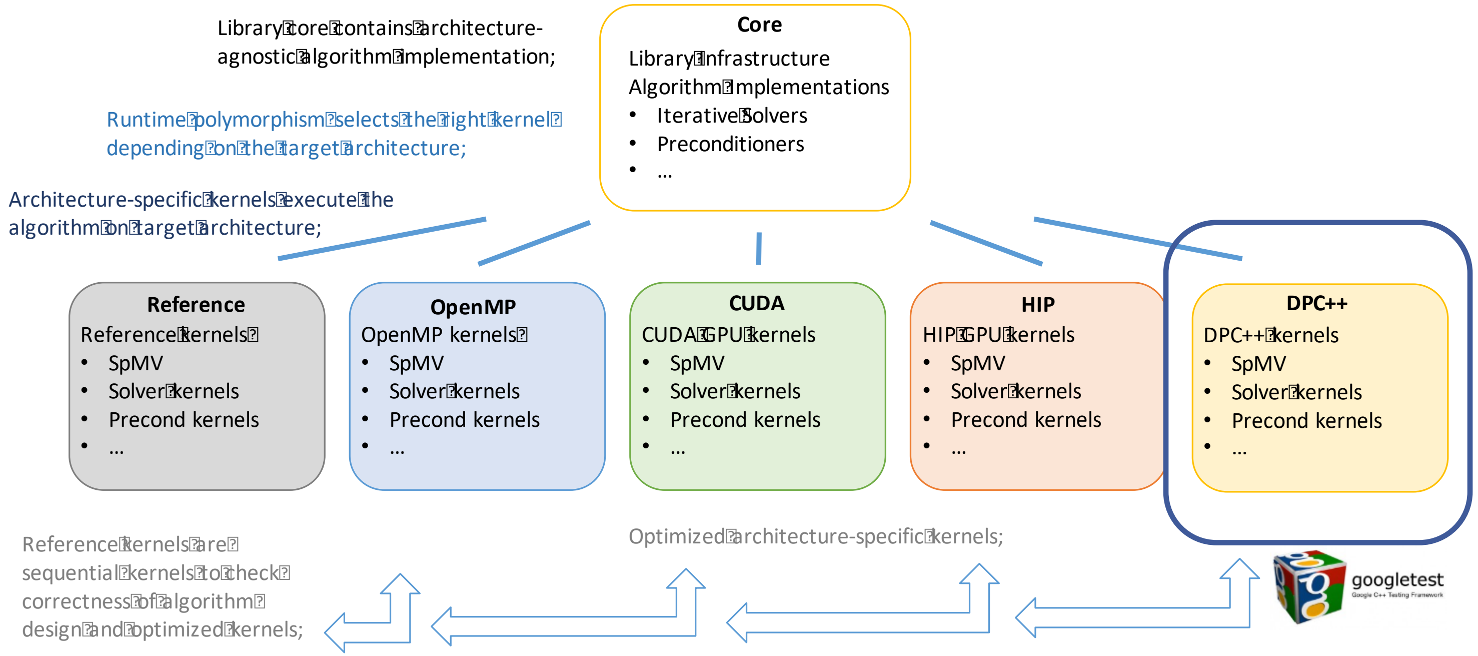
Allocate(size, …)
Free(ptr)
Copy(ptr, OtherExec)
…

Dpc++Executor

Allocate(size, …)
Free(ptr)
Copy(ptr, OtherExec)
…

Porting Ginkgo's CUDA Code to the oneAPI Ecosystem

Ginkgo



Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

Architecture-specific kernels execute the algorithm on target architecture;

Core
Library Infrastructure
Algorithm Implementations
- Iterative Solvers
- Preconditioners
- …

Reference
Reference kernels
- SpMV
- Solver kernels
- Precond kernels
- …

OpenMP
OpenMP kernels
- SpMV
- Solver kernels
- Precond kernels
- …

CUDA
CUDA GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

HIP
HIP GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

DPC++
DPC++ kernels
- SpMV
- Solver kernels
- Precond kernels
- …

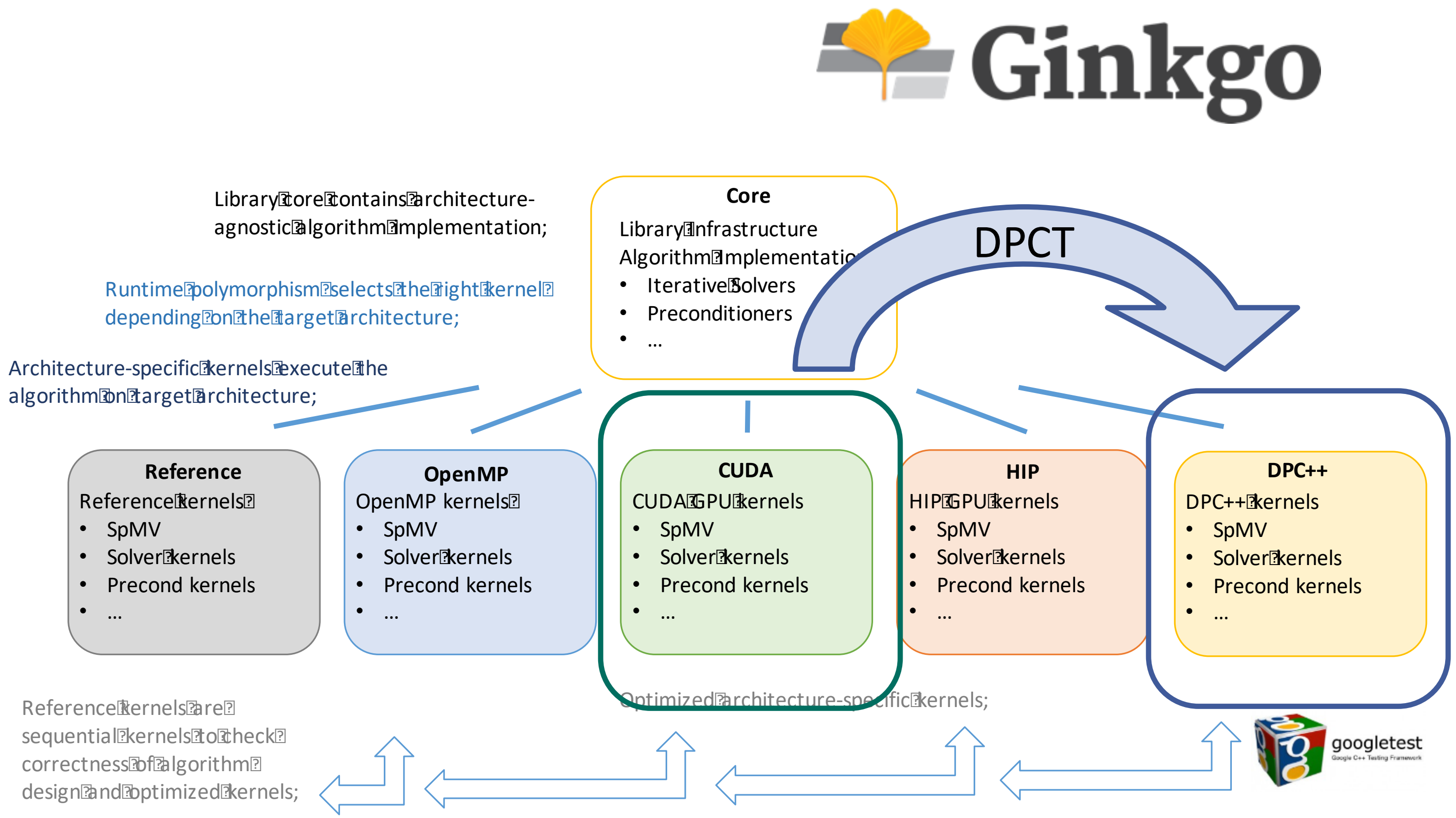
Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;

Optimized architecture-specific kernels;

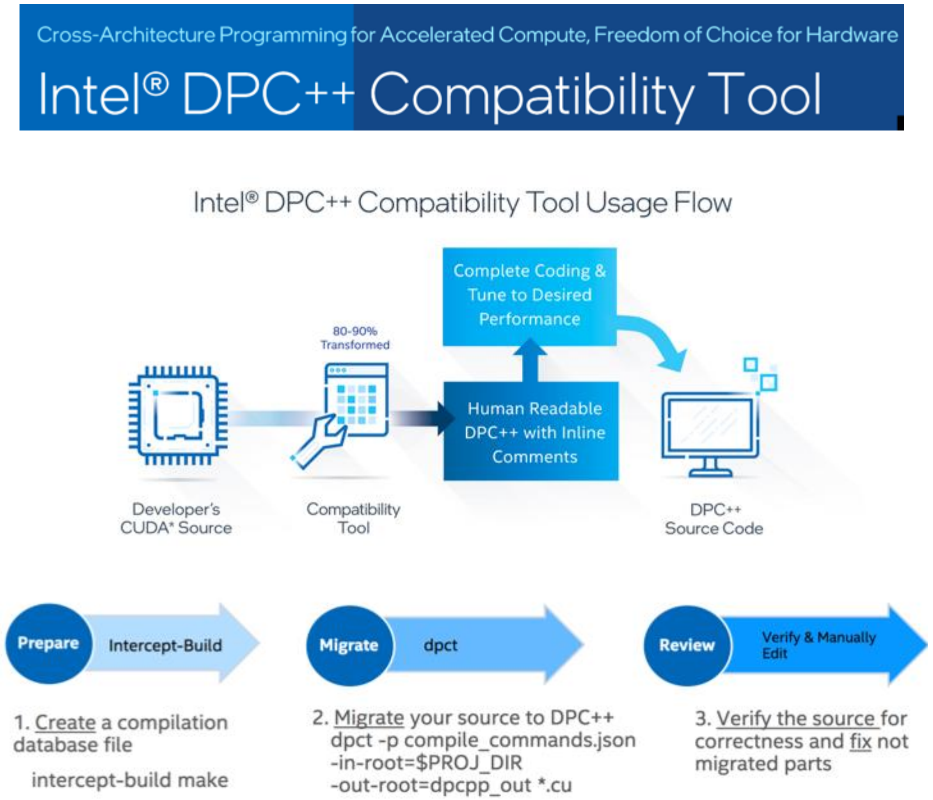
Three steps to porting:

1. Add a **new executor** and other core library infrastructure and **new stub kernels**. Also add key **kernel programming components** (cooperative group, reductions, …). **Semi-Manual.**
    1. DPC++ USM
    2. DPC++ in_order queues persistent for the executor
2. **Automatically** port kernels one by one using DPCT.
3. Tune performance and leverage advanced features. **Manual.**

Porting Ginkgo’s CUDA Code to the oneAPI Ecosystem



We generate the Ginkgo DPC++ backend from the CUDA backend via DPCT porting tool.


Cross-Architecture Programming for Accelerated Compute, Freedom of Choice for Hardware
Intel® DPC++ Compatibility Tool
Intel® DPC++ Compatibility Tool Usage Flow
Complete Coding & Tune to Desired Performance
80-90% Transformed
Human Readable DPC++ with Inline Comments
Developer's CUDA* Source
Compatibility Tool
DPC++ Source Code
Prepare
Intercept-Build
Migrate
dpct
Review
Verify & Manually Edit

1. Create a compilation database file

intercept-build make

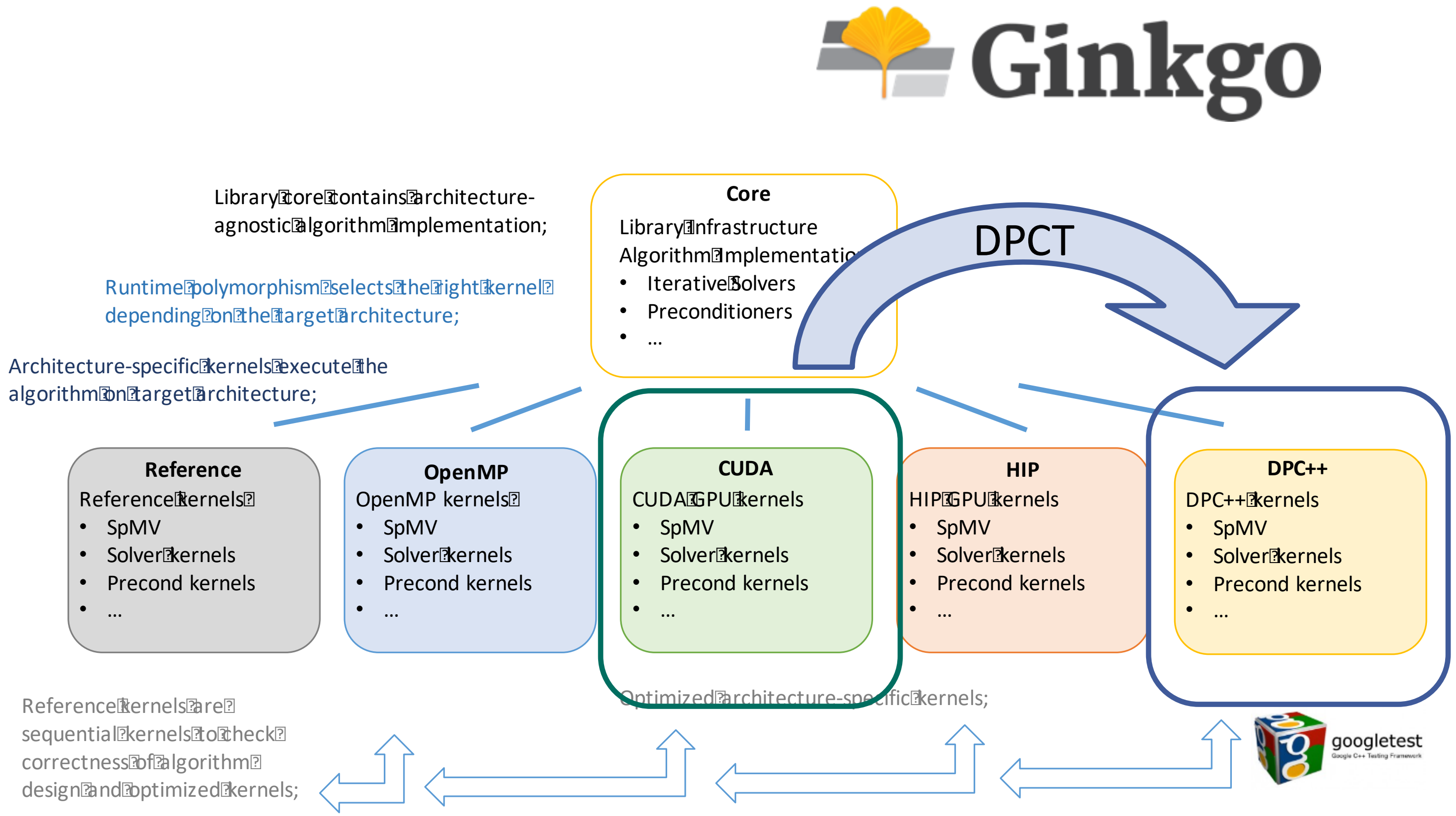
2. Migrate your source to DPC++

dpct -p compile_commands.json -in-root=$PROJ_DIR -out-root=dpcpp_out *.cu

3. Verify the source for correctness and fix not migrated parts

https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-intel-dpcpp-compatibility-tool/top.html

Porting Ginkgo's CUDA Code to the oneAPI Ecosystem



Ginkgo

Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

Architecture-specific kernels execute the algorithm on target architecture;

Core
Library Infrastructure
Algorithm Implementatio
- Iterative Solvers
- Preconditioners
- …

DPCT

Reference
Reference kernels
- SpMV
- Solver kernels
- Precond kernels
- …

OpenMP
OpenMP kernels
- SpMV
- Solver kernels
- Precond kernels
- …

CUDA
CUDA GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

HIP
HIP GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

DPC++
DPC++ kernels
- SpMV
- Solver kernels
- Precond kernels
- …

Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;

Optimized architecture-specific kernels;

googletest

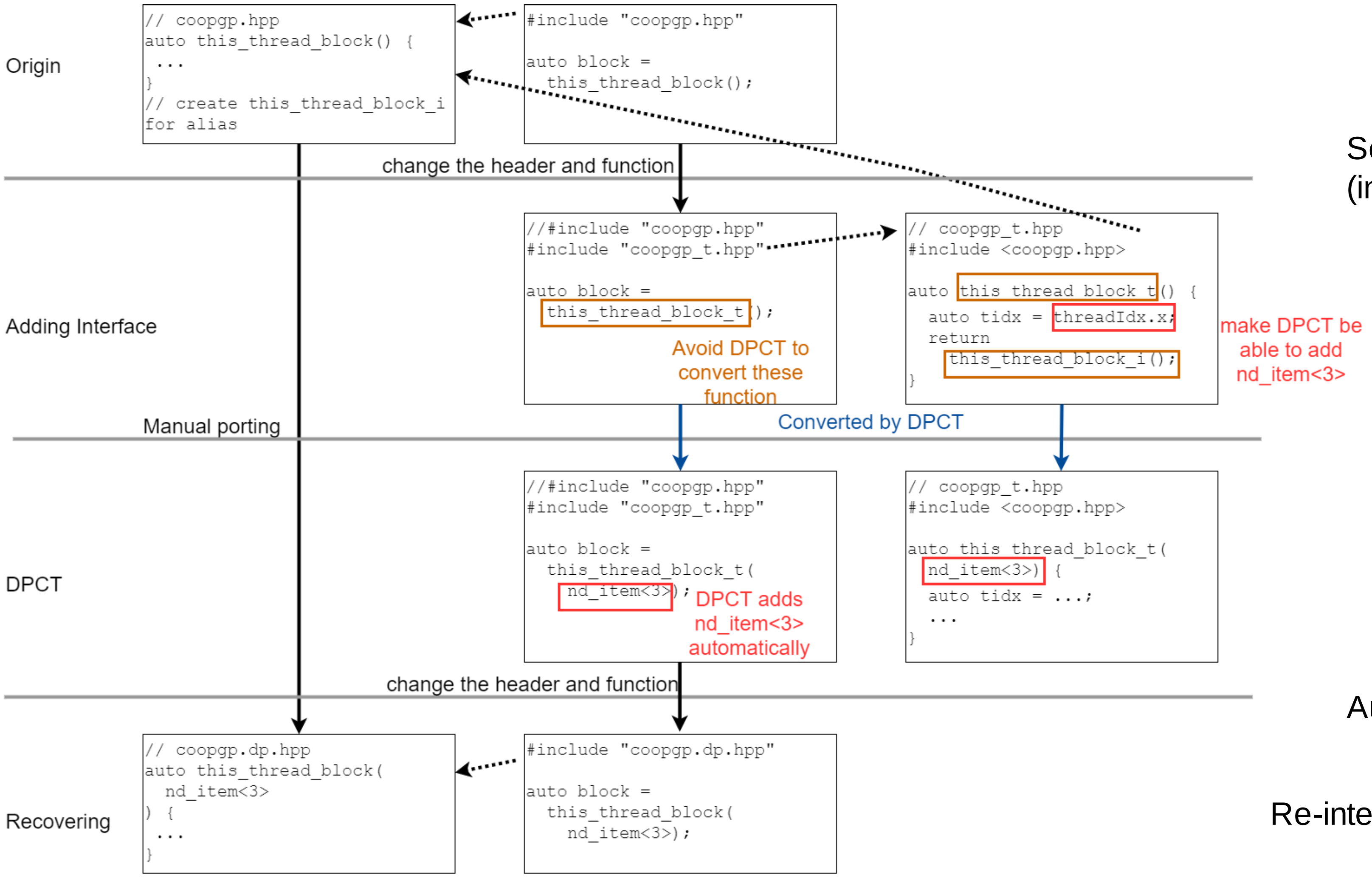
We generate the Ginkgo DPC++ backend from the CUDA backend via DPCT porting tool.

Encountered Limitations:

- DPCT requires knowledge about all functionality dependencies -> need to mimic dependencies;
- The conversion fails for heavily-templated functionality;
- DPCT fails to convert cooperative group functionality or atomics, both needed for high performance;
- Ginkgo's software architecture requires some customized solutions;
- We want to run on diverse Intel oneAPI architectures;

https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-intel-dpcpp-compatibility-tool/top.html

Porting Ginkgo's CUDA Code to the oneAPI Ecosystem


Origin
Adding Interface
DPCT
Recovering
change the header and function
Manual porting
Converted by DPCT
change the header and function
Avoid DPCT to convert these function
make DPCT be able to add nd_item<3>
DPCT adds nd_item<3> automatically

Origin:

```
// coopgp.hpp
auto this_thread_block() {
  ...
}
// create this_thread_block_i
for alias
```

```
#include "coopgp.hpp"

auto block =
  this_thread_block();
```

Adding Interface:

```
//#include "coopgp.hpp"
#include "coopgp_t.hpp"

auto block =
  this_thread_block_t();
```

```
// coopgp_t.hpp
#include <coopgp.hpp>

auto this_thread_block_t() {
  auto tidx = threadIdx.x;
  return
  this_thread_block_i();
}
```

DPCT:

```
//#include "coopgp.hpp"
#include "coopgp_t.hpp"

auto block =
  this_thread_block_t(
    nd_item<3>);
```

```
// coopgp_t.hpp
#include <coopgp.hpp>

auto this_thread_block_t(
  nd_item<3>) {
  auto tidx = ...;
  ...
}
```

Recovering:

```
// coopgp.dp.hpp
auto this_thread_block(
  nd_item<3>
) {
  ...
}
```

```
#include "coopgp.dp.hpp"

auto block =
  this_thread_block(
    nd_item<3>);
```

Separate functions to convert (including necessary headers);

Employ workarounds preventing DPCT from invalid conversions;

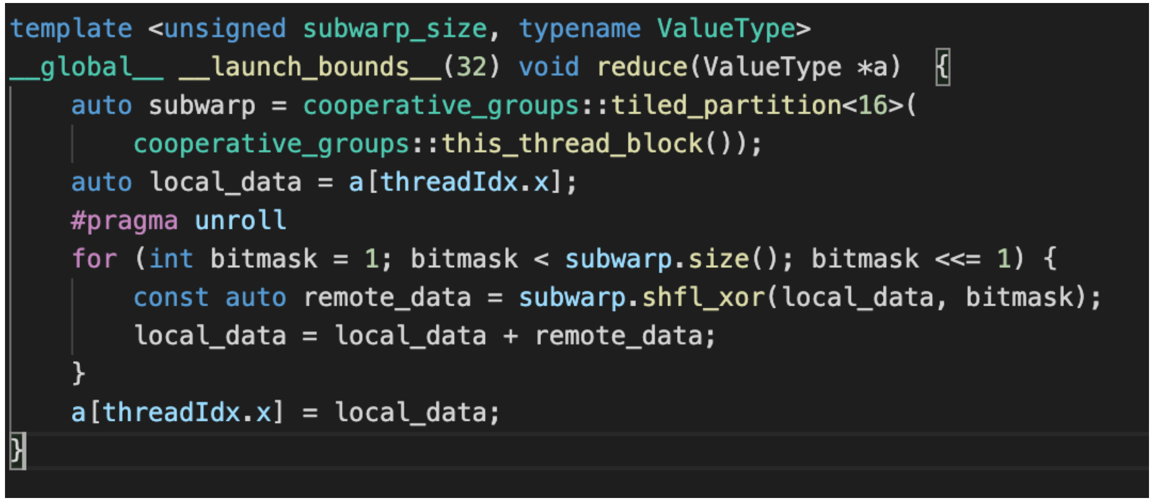
Run DPCT on isolated kernels;

Automatically apply additional fixes;

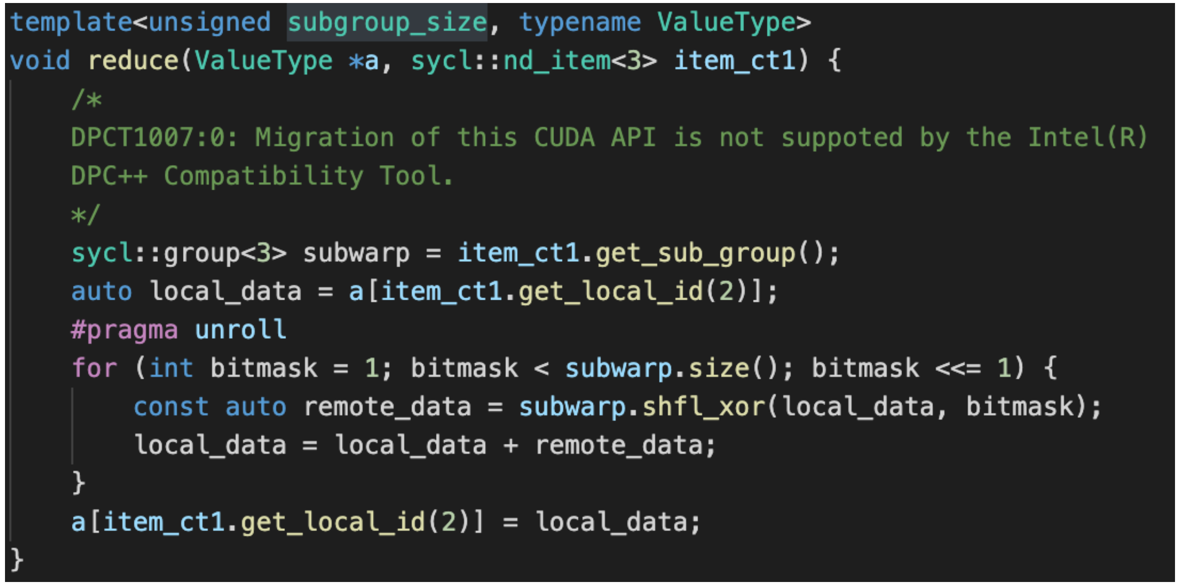
Re-integrate into software stack

Re-engineering cooperative group calls

DPCT fails to handle cooperative group statements



```
template <unsigned subwarp_size, typename ValueType>
__global__ __launch_bounds__(32) void reduce(ValueType *a)  {
    auto subwarp = cooperative_groups::tiled_partition<16>(
        cooperative_groups::this_thread_block());
    auto local_data = a[threadIdx.x];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[threadIdx.x] = local_data;
}
```

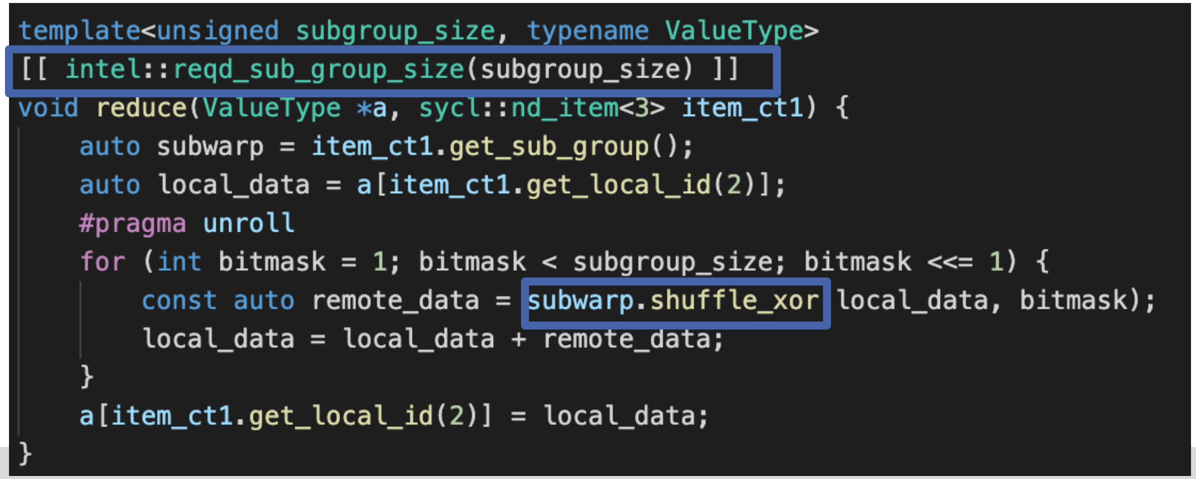


```
template<unsigned subgroup_size, typename ValueType>
void reduce(ValueType *a, sycl::nd_item<3> item_ct1) {
    /*
    DPCT1007:0: Migration of this CUDA API is not suppoted by the Intel(R)
    DPC++ Compatibility Tool.
    */
    sycl::group<3> subwarp = item_ct1.get_sub_group();
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```

Different way to assign the group size

Manually insert cooperative group calls

Different name for "shuffle"



```
template<unsigned subgroup_size, typename ValueType>
[[ intel::reqd_sub_group_size(subgroup_size) ]]
void reduce(ValueType *a, sycl::nd_item<3> item_ct1) {
    auto subwarp = item_ct1.get_sub_group();
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subgroup_size; bitmask <<= 1) {
        const auto remote_data = subwarp.shuffle_xor local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```

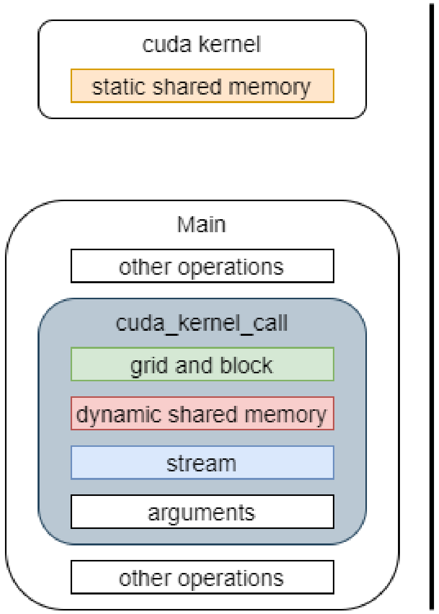
Aim for similar backend designs

CUDA

- In-kernel static shared memory allocation
- In-sync execution

DPC++

- Static shared memory allocated outside kernel
- SYCL-style asynchronous kernel execution



Aim for similar backend designs

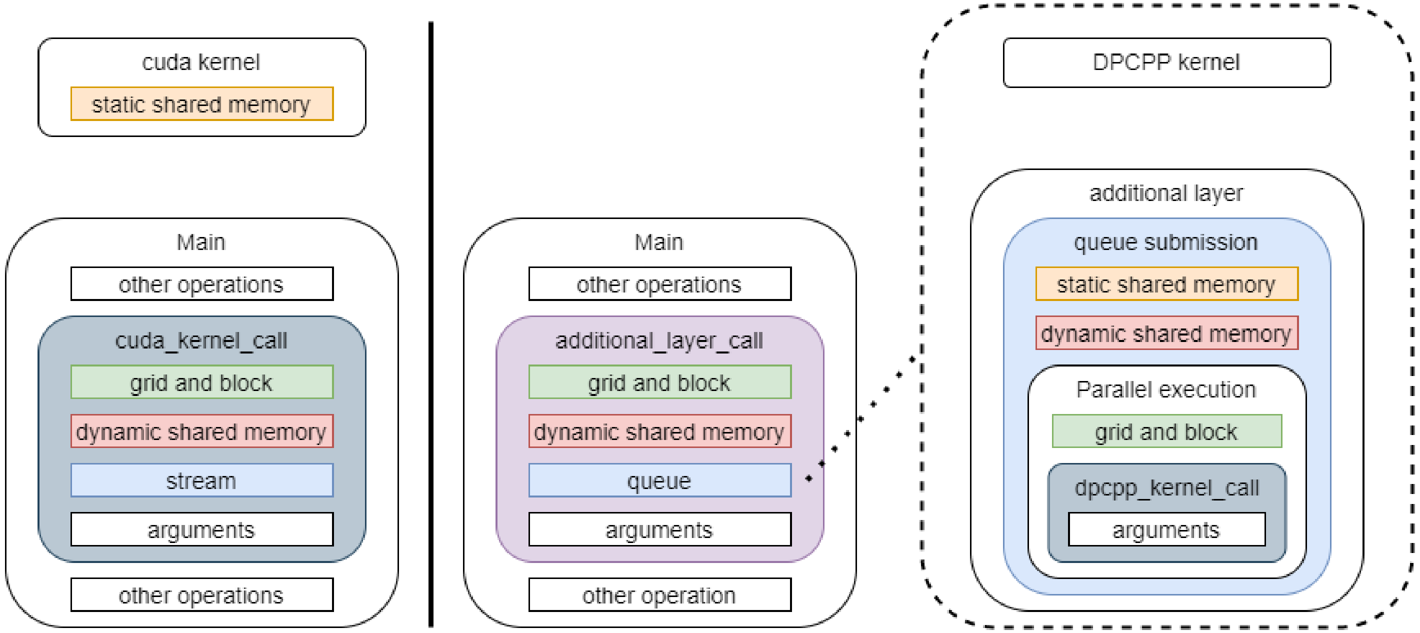
CUDA

- In-kernel static shared memory allocation
- In-sync execution

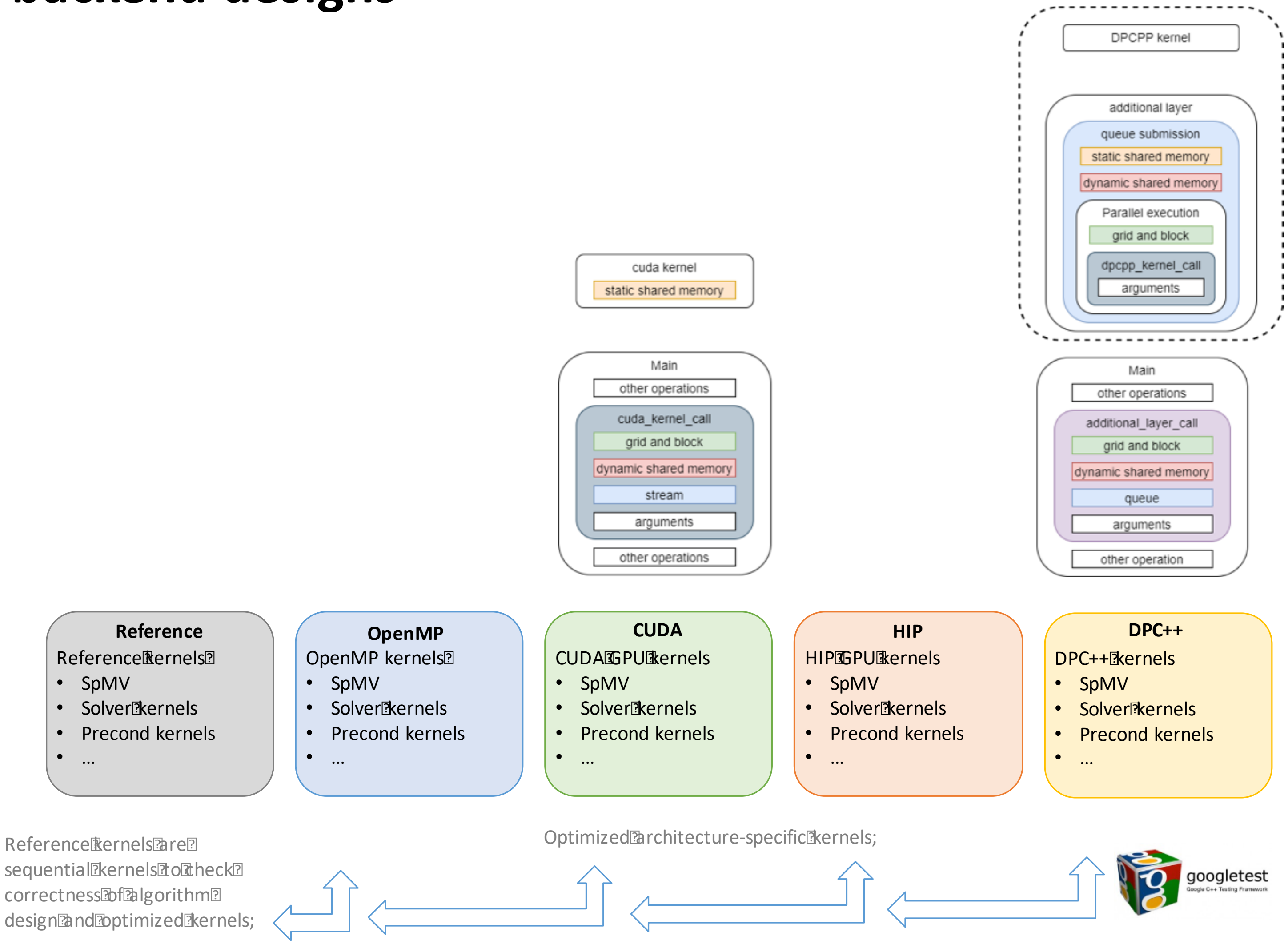
DPC++

- Static shared memory allocated outside kernel
- SYCL-style asynchronous kernel execution

➢ Mimic backend similarity by adding intermediate layer
➢ Synchronize streams
➢ Handle static shared memory allocation in intermediate layer



Aim for similar backend designs



cuda kernel
static shared memory

Main
other operations
cuda_kernel_call
grid and block
dynamic shared memory
stream
arguments
other operations

DPCPP kernel
additional layer
queue submission
static shared memory
dynamic shared memory
Parallel execution
grid and block
dpcpp_kernel_call
arguments

Main
other operations
additional_layer_call
grid and block
dynamic shared memory
queue
arguments
other operation

Reference
Reference kernels
- SpMV
- Solver kernels
- Precond kernels
- …

OpenMP
OpenMP kernels
- SpMV
- Solver kernels
- Precond kernels
- …

CUDA
CUDA GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

HIP
HIP GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

DPC++
DPC++ kernels
- SpMV
- Solver kernels
- Precond kernels
- …

Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;

Optimized architecture-specific kernels;

The oneAPI hardware zoo

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- *Intel GPUs support subgroup sizes 8, 16, 32*
- *Intel CPUs support subgroup sizes 4, 8, (16)*
- *Gen9 GPUs support max workgroup size 256*
- *DG1 GPUs support max workgroup size 512*
- …

➢ We need a way to choose a valid (and good!) configuration for all hardware architectures.

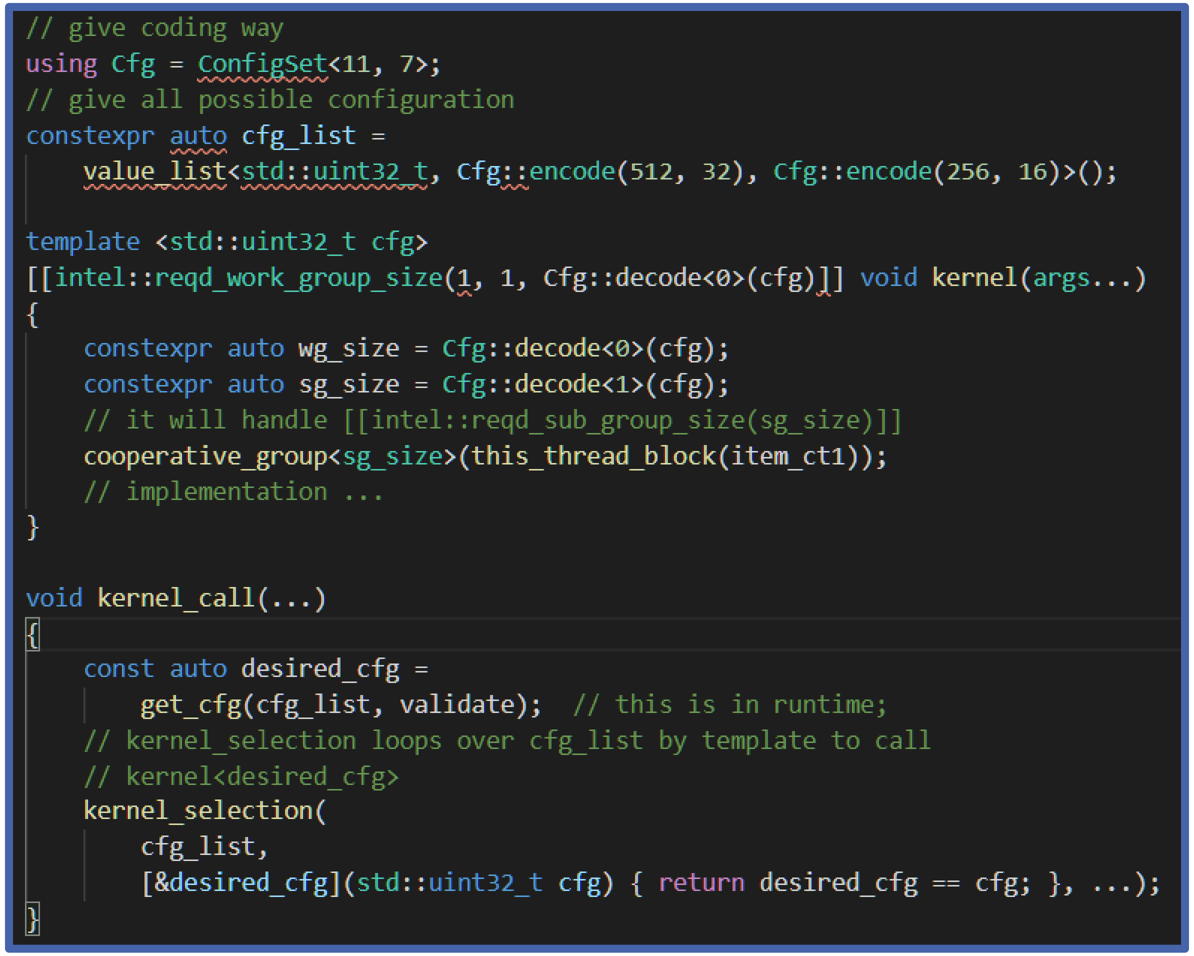
The oneAPI hardware zoo

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- *Intel GPUs support subgroup sizes 8, 16, 32*
- *Intel CPUs support subgroup sizes 4, 8, (16)*
- *Gen9 GPUs support max workgroup size 256*
- *DG1 GPUs support max workgroup size 512*
- …

➢ We need a way to choose a valid (and good!) configuration for all hardware architectures.

We use the ConfigSet to encode all info into one number and couple this concept with generating the kernels for all valid configs at compile time and selecting the kernel suitable for the given hardware at compile time.

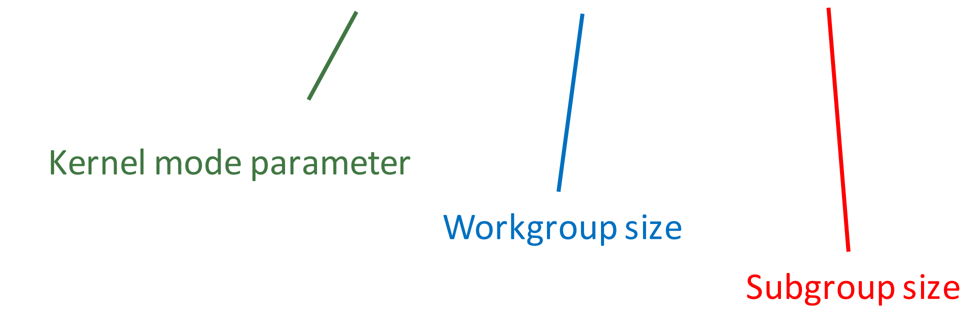


```
// give coding way
using Cfg = ConfigSet<11, 7>;
// give all possible configuration
constexpr auto cfg_list =
    value_list<std::uint32_t, Cfg::encode(512, 32), Cfg::encode(256, 16)>();

template <std::uint32_t cfg>
[[intel::reqd_work_group_size(1, 1, Cfg::decode<0>(cfg))]] void kernel(args...)
{
    constexpr auto wg_size = Cfg::decode<0>(cfg);
    constexpr auto sg_size = Cfg::decode<1>(cfg);
    // it will handle [[intel::reqd_sub_group_size(sg_size)]]
    cooperative_group<sg_size>(this_thread_block(item_ct1));
    // implementation ...
}

void kernel_call(...)
{
    const auto desired_cfg =
        get_cfg(cfg_list, validate);  // this is in runtime;
    // kernel_selection loops over cfg_list by template to call
    // kernel<desired_cfg>
    kernel_selection(
        cfg_list,
        [&desired_cfg](std::uint32_t cfg) { return desired_cfg == cfg; }, ...);
}
```

The ConfigSet encoding

We store the ConfigSet as bits for a bitmask comparison:

use 32 bits	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
encoding for ConfigSet<3, 11, 7>	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
encode the config encode(4, 256, 16)	4 256 16
the result is 4 * 2^18 + 256 * 2^7 + 16	00000000000010001000000000010000



Kernel mode parameter

Workgroup size

Subgroup size

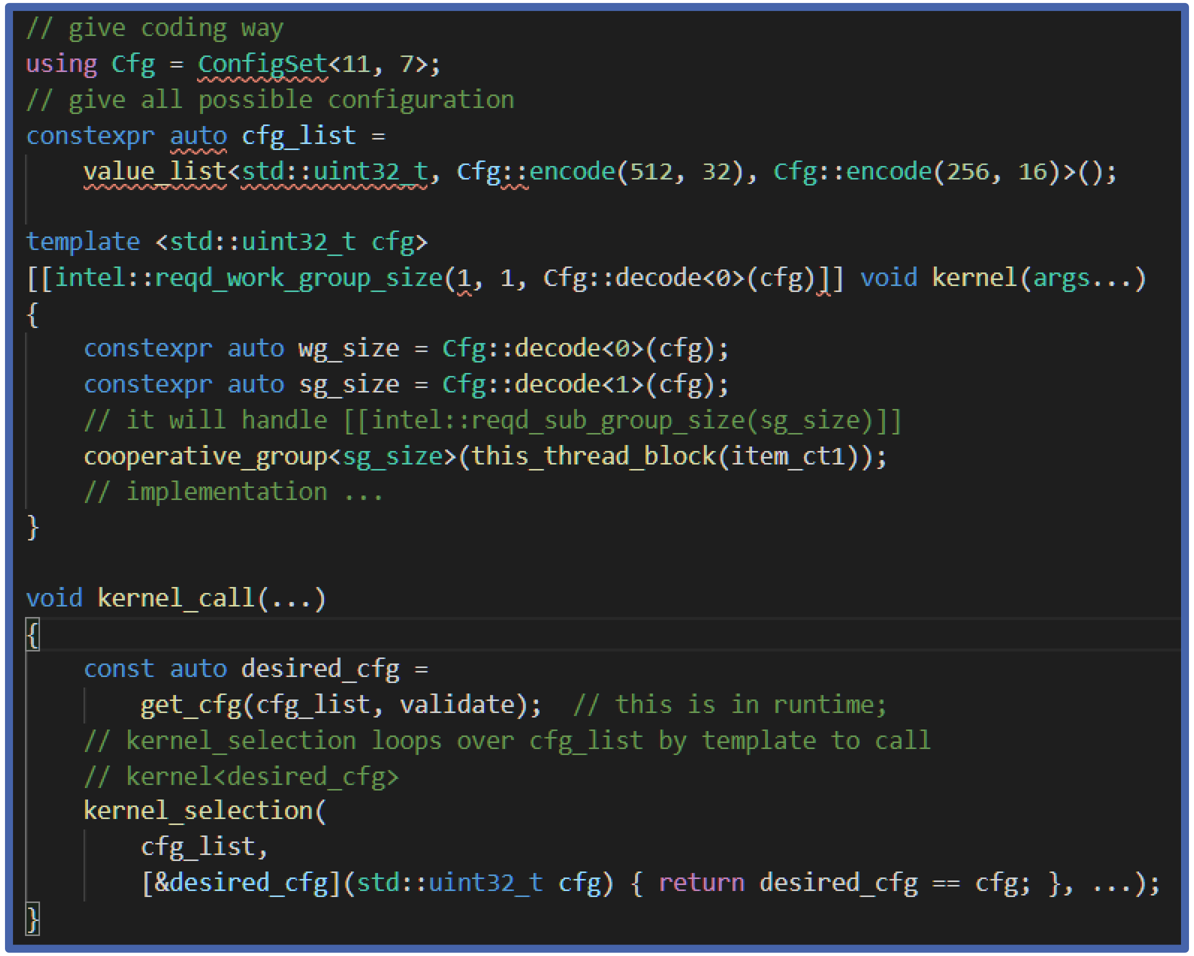
The ConfigSet encoding

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- *Intel GPUs support subgroup sizes 8, 16, 32*
- *Intel CPUs support subgroup sizes 4, 8, (16)*
- *Gen9 GPUs support max workgroup size 256*
- *DG1 GPUs support max workgroup size 512*
- …

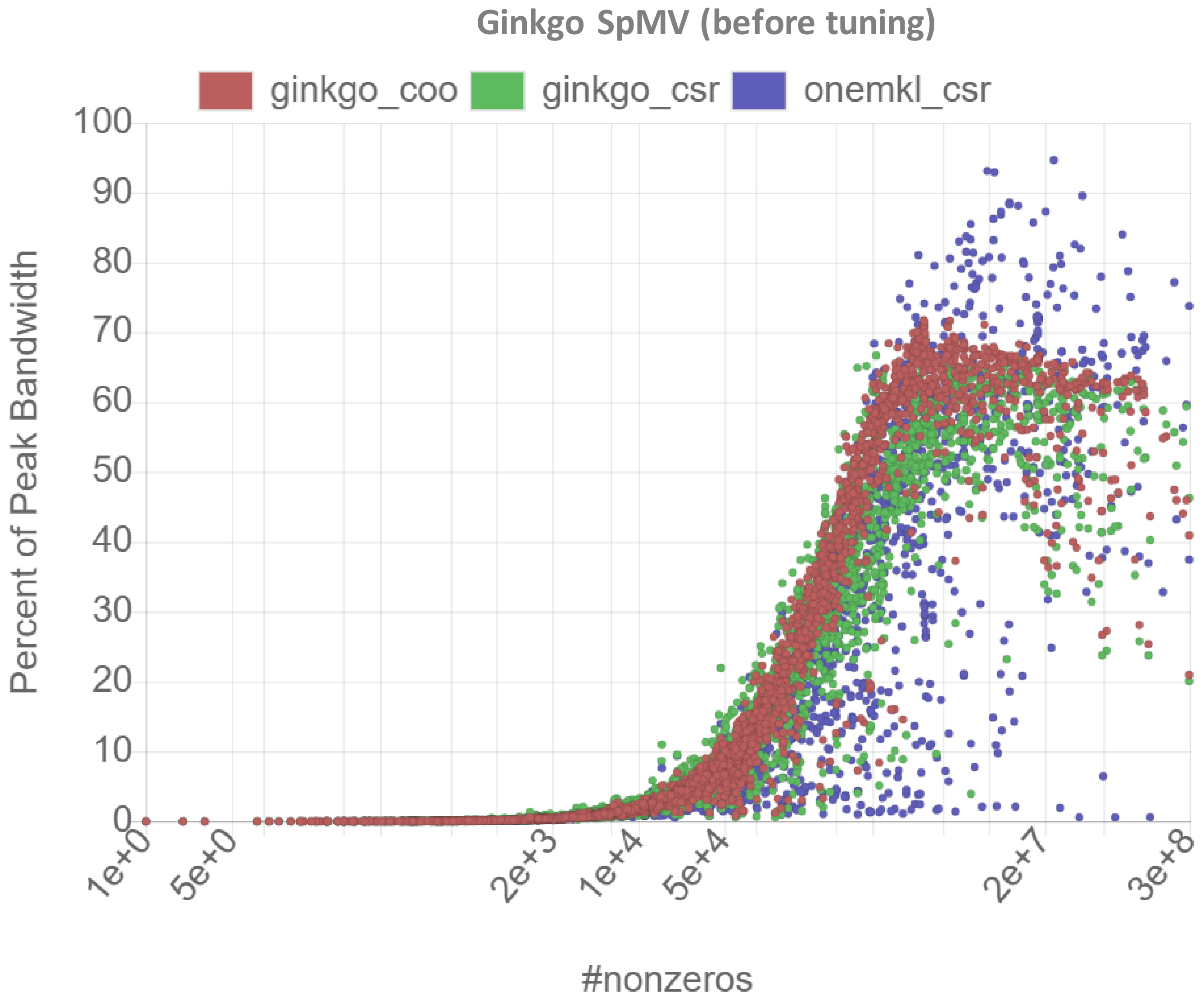
➢ We need a way to choose a valid (and good!) configuration for all hardware architectures.

We use the ConfigSet to encode all info into one number and couple this concept with generating the kernels for all valid configs at compile time and selecting the kernel suitable for the given hardware at compile time.



```
// give coding way
using Cfg = ConfigSet<11, 7>;
// give all possible configuration
constexpr auto cfg_list =
    value_list<std::uint32_t, Cfg::encode(512, 32), Cfg::encode(256, 16)>();

template <std::uint32_t cfg>
[[intel::reqd_work_group_size(1, 1, Cfg::decode<0>(cfg)]] void kernel(args...)
{
    constexpr auto wg_size = Cfg::decode<0>(cfg);
    constexpr auto sg_size = Cfg::decode<1>(cfg);
    // it will handle [[intel::reqd_sub_group_size(sg_size)]]
    cooperative_group<sg_size>(this_thread_block(item_ct1));
    // implementation ...
}

void kernel_call(...)
{
    const auto desired_cfg =
        get_cfg(cfg_list, validate);  // this is in runtime;
    // kernel_selection loops over cfg_list by template to call
    // kernel<desired_cfg>
    kernel_selection(
        cfg_list,
        [&desired_cfg](std::uint32_t cfg) { return desired_cfg == cfg; }, ...);
}
```

Ginkgo's Functionality and Performance in the DPC++ ecosystem

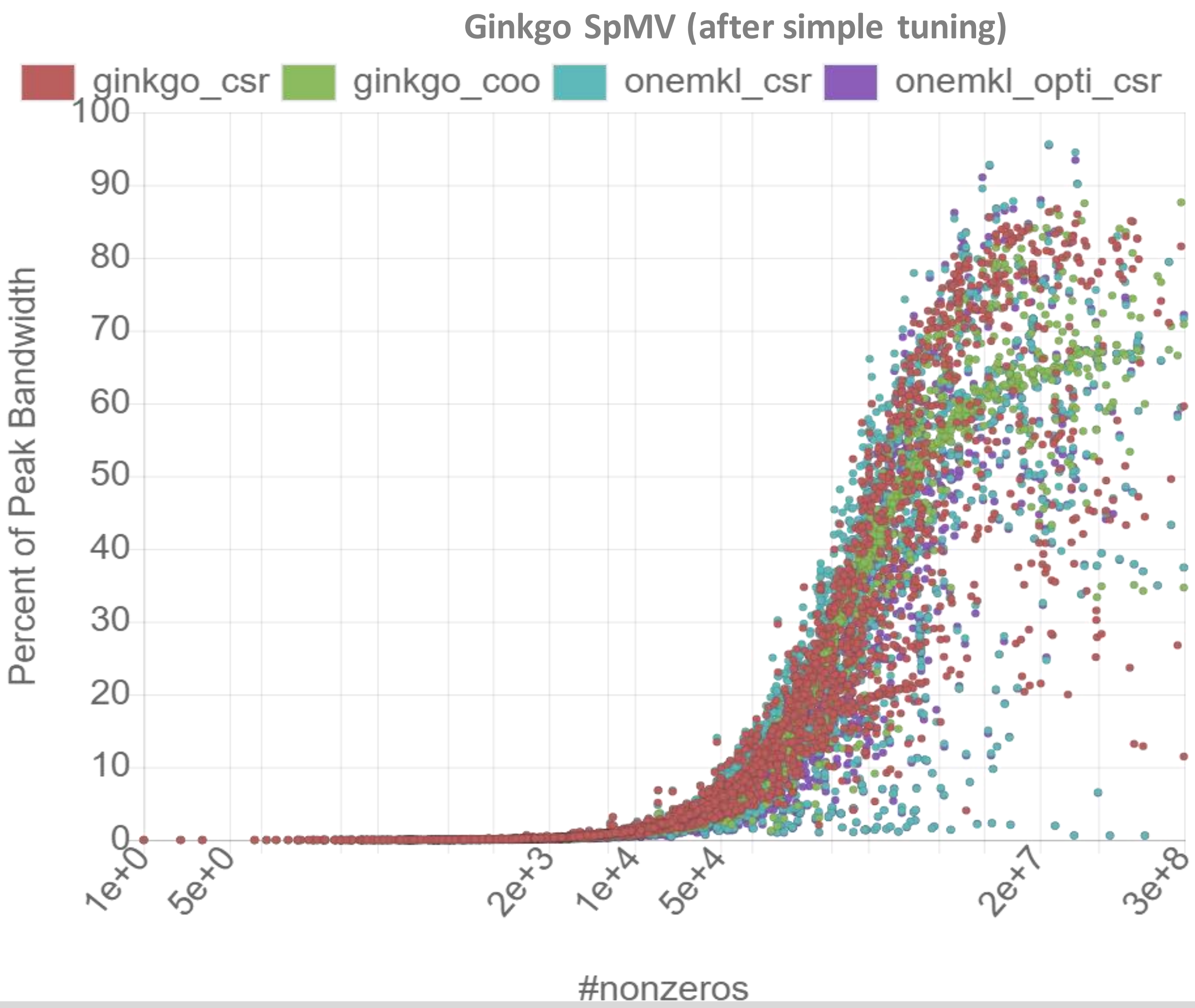


Performance of Ginkgo's SpMV kernels and OneMKL SpMV on a Intel DG1 GPU (float).

Test case: (All) Matrices available in the Suite Sparse Matrix Collection* that fit the GPU memory.

*https://people.engr.tamu.edu/davis/suitesparse.html

Ginkgo's Functionality and Performance in the DPC++ ecosystem

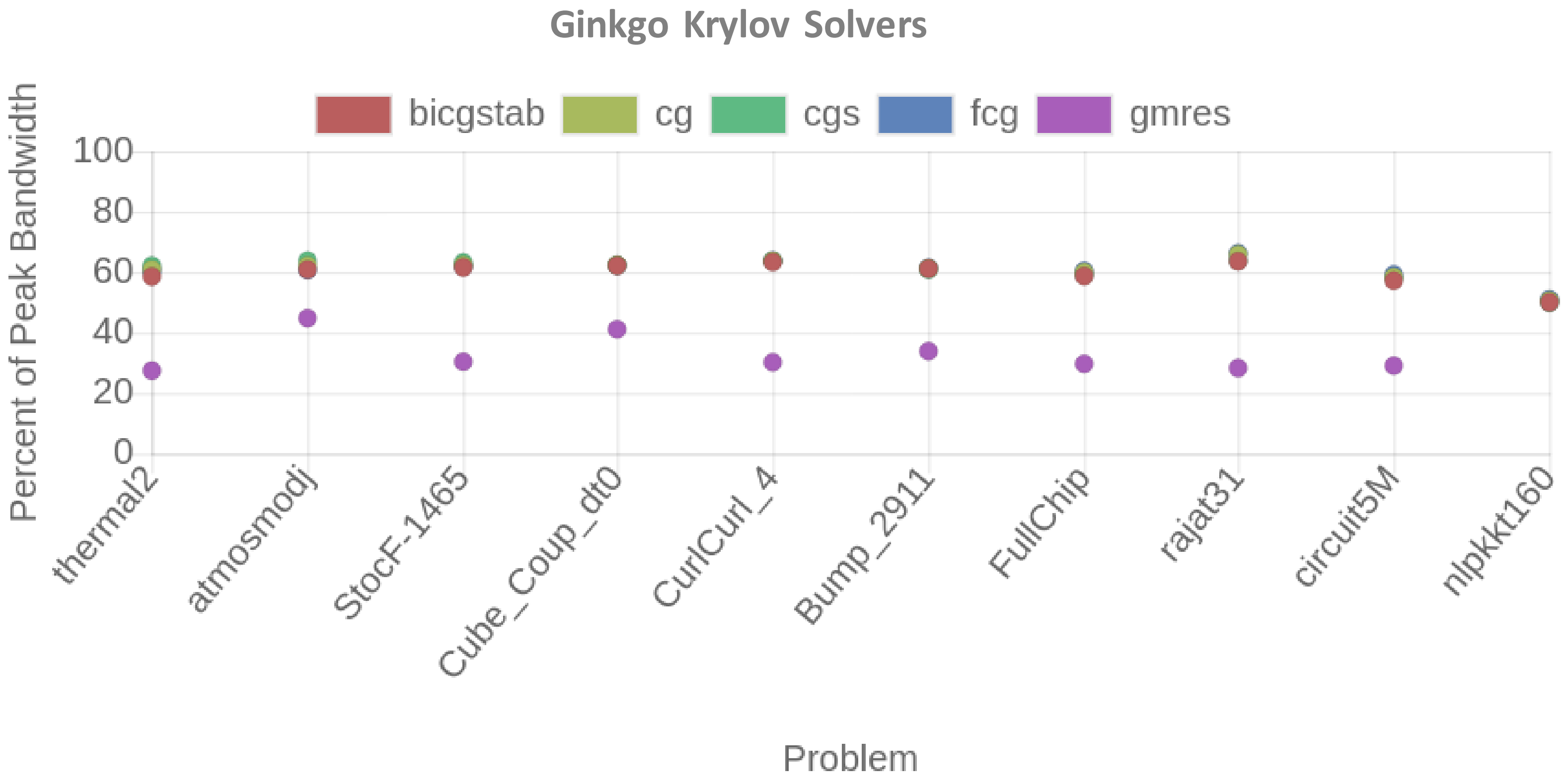


Performance of Ginkgo's SpMV kernels and OneMKL SpMV on a Intel DG1 GPU (float).

Test case: (All) Matrices available in the Suite Sparse Matrix Collection* that fit the GPU memory.

*https://people.engr.tamu.edu/davis/suitesparse.html

Ginkgo’s Functionality and Performance in the DPC++ ecosystem



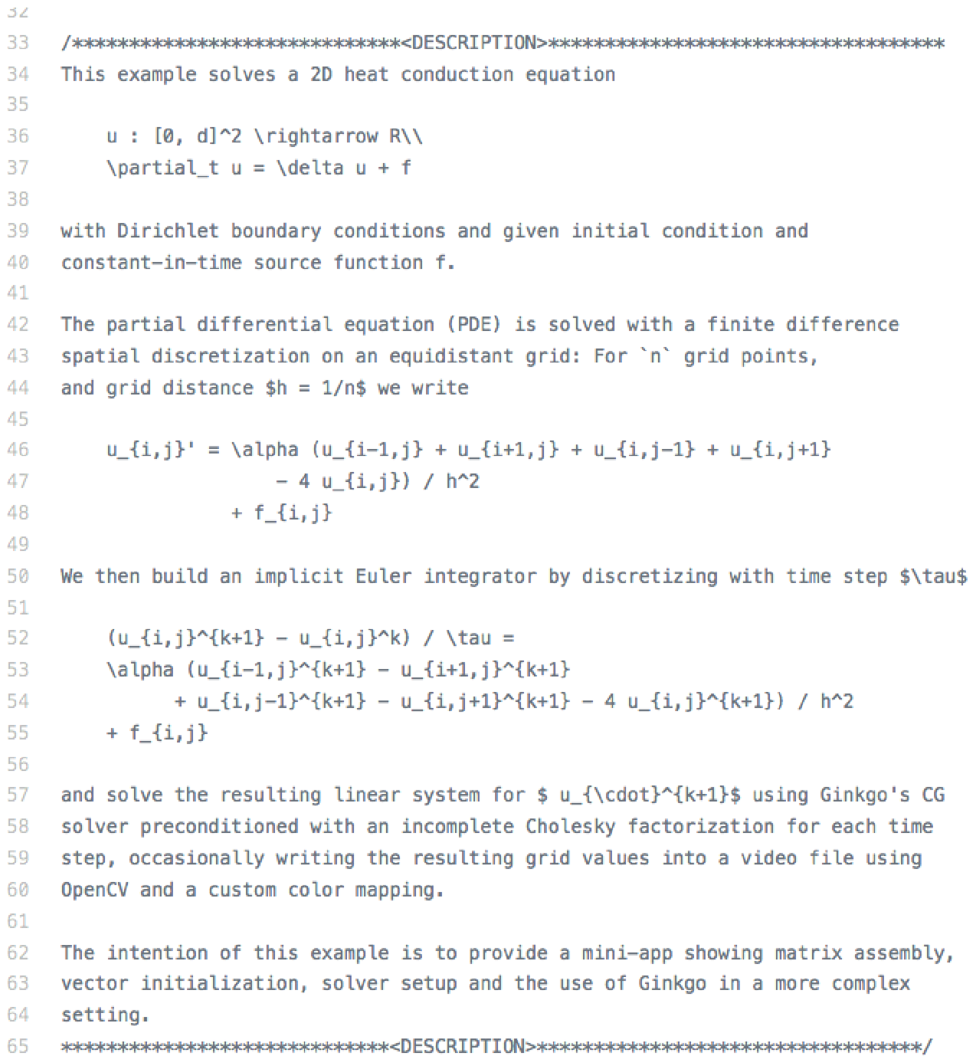
Performance of Ginkgo’s Krylov solver on an Intel Gen9 GPU.

*https://people.engr.tamu.edu/davis/suitesparse.html

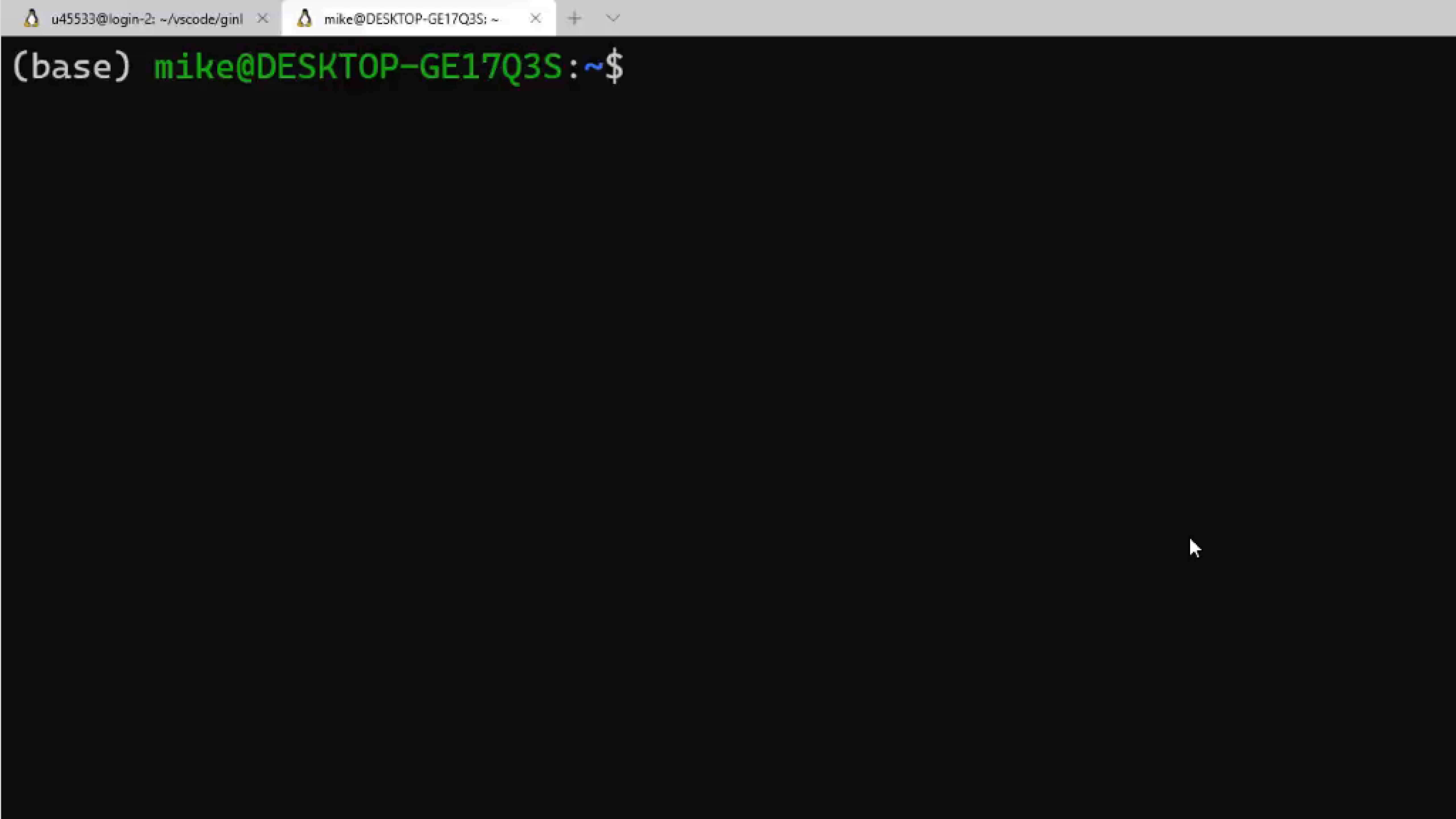
Using Ginkgo’s DPC++ backend to run simulations in the oneAPI ecosystem

https://github.com/ginkgo-project/ginkgo/blob/develop/examples/heat-equation/heat-equation.cpp

Video realizing the heat equation simulation in the Intel DevCloud:
http://www.icl.utk.edu/~hanzt/slides/GinkgoHeatEqDevCloud.mp4



```
/*****************************<DESCRIPTION>**********************************
This example solves a 2D heat conduction equation

    u : [0, d]^2 \rightarrow R\\
    \partial_t u = \delta u + f

with Dirichlet boundary conditions and given initial condition and
constant-in-time source function f.

The partial differential equation (PDE) is solved with a finite difference
spatial discretization on an equidistant grid: For `n` grid points,
and grid distance $h = 1/n$ we write

    u_{i,j}' = \alpha (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}
                   - 4 u_{i,j}) / h^2
               + f_{i,j}

We then build an implicit Euler integrator by discretizing with time step $\tau$

    (u_{i,j}^{k+1} - u_{i,j}^k) / \tau =
    \alpha (u_{i-1,j}^{k+1} - u_{i+1,j}^{k+1}
          + u_{i,j-1}^{k+1} - u_{i,j+1}^{k+1} - 4 u_{i,j}^{k+1}) / h^2
    + f_{i,j}

and solve the resulting linear system for $ u_{\cdot}^{k+1}$ using Ginkgo's CG
solver preconditioned with an incomplete Cholesky factorization for each time
step, occasionally writing the resulting grid values into a video file using
OpenCV and a custom color mapping.

The intention of this example is to provide a mini-app showing matrix assembly,
vector initialization, solver setup and the use of Ginkgo in a more complex
setting.
*****************************<DESCRIPTION>********************************/
```



```
(base) mike@DESKTOP-GE17Q3S:~$
```

Next Steps

Porting completed for:

- Basic BLAS functionality
- Sparse matrix vector product
  - CSR, COO, ELL, SellP, hybrid
- Iterative linear solvers
  - CG, BiCGSTAB, CGS, FCG, GMRES
- Simulation workflows

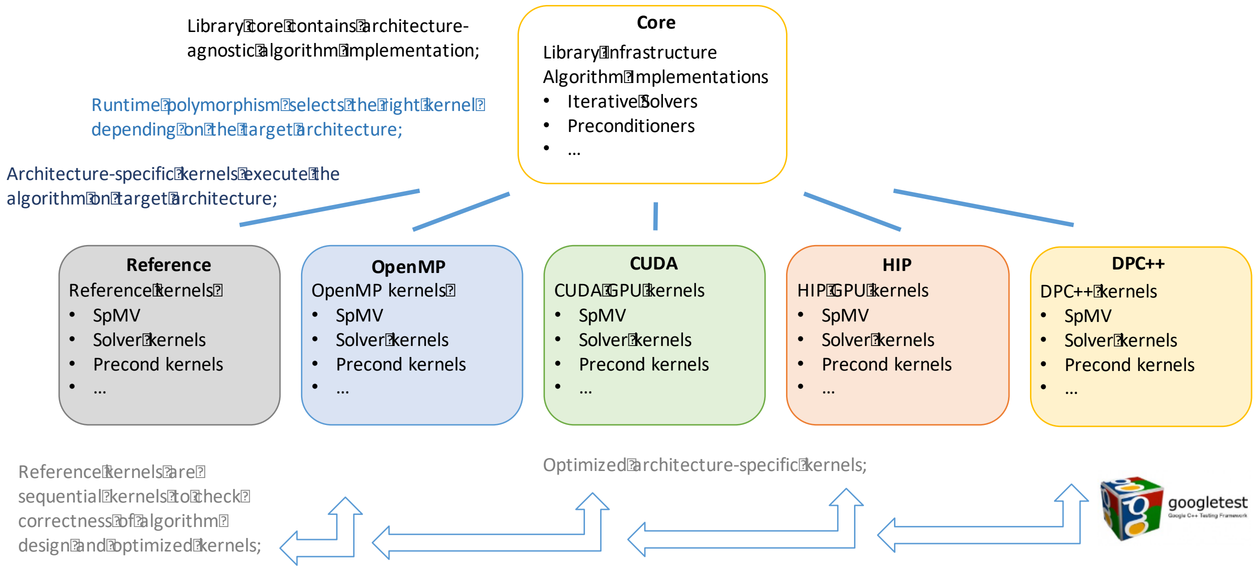
Porting ongoing for:

- SpGEMM, SpGEAM
- IDR Krylov solver
- Multigrid methods
- Advanced preconditioners (ParILU, ParILUT, block-Jacobi…)

- Evaluate performance of DPC++ backend on AMD and NVIDIA devices;

Ginkgo

GPU-centric high performance sparse linear algebra ecosystem. Sustainable and extensible ecosystem with support for AMD GPUs, NVIDIA GPUs, and Intel GPUs.



Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

Architecture-specific kernels execute the algorithm on target architecture;

Core
Library infrastructure
Algorithm implementations
- Iterative Solvers
- Preconditioners
- …

Reference
Reference kernels
- SpMV
- Solver kernels
- Precond kernels
- …

OpenMP
OpenMP kernels
- SpMV
- Solver kernels
- Precond kernels
- …

CUDA
CUDA GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

HIP
HIP GPU kernels
- SpMV
- Solver kernels
- Precond kernels
- …

DPC++
DPC++ kernels
- SpMV
- Solver kernels
- Precond kernels
- …

Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;

Optimized architecture-specific kernels;

googletest

We want to thank the Intel Development team for all the help we receive!

In particular, Alina Shadrina, George Silva, Sujata Tibrewala, Klaus-Dieter Oertel, Edmund Preiss, and Arti Gupta.